Chapter I

Using Generative Representations to Evolve Robots

Gregory S. Hornby [I.1]

Recent research has demonstrated the ability of evolutionary algorithms to automatically design both the physical structure and software controller of real physical robots. One of the challenges for these automated design systems is to improve their ability to scale to the high complexities found in real-world problems. Here we claim that for automated design systems to scale in complexity they must use a representation which allows for the hierarchical creation and reuse of modules, which we call a *generative representation*. Not only is the ability to reuse modules necessary for functional scalability, but it is also valuable for improving efficiency in testing and construction. We then describe an evolutionary design system with a generative representation capable of hierarchical modularity and demonstrate it for the design of locomoting robots in simulation. Finally, results from our experiments show that evolution with our generative representation produces better robots than those evolved with a non-generative representation.

I.1 Introduction

Computer-automated design systems using evolutionary algorithms have been used to evolve a variety of static objects – such as antennas [11], load cells [16], trusses [14] and more [1, 2] – software controllers for robots [15, 20, 3, 6], and both the controller and structure of robots [17, 21, 10, 12]. For the most part the designs

[I.1] QSS Group Inc., NASA Ames Research Center, Mail Stop 269-3, Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov, http://ic.arc.nasa.gov/people/hornby/

that have been produced are fairly simple, which has led to a concern with how well this method will scale to the high complexities necessary for real-world robot designs. In engineering and software development, complex artifacts are achieved by exploiting the principles of regularity, modularity, hierarchy and reuse [19] [8] [13]. These features can be summarized as the hierarchical reuse of building blocks.

Here we claim that for automatic design systems to scale in complexity the designs they produce must be made of reused modules. While the evolutionary algorithm can affect the degree of reuse in an evolved design, the ability to create structures which reuse subassemblies is limited by the ability of the representation to encode them. We define *generative representations* as the class of representations which allow building-blocks encoded in the genotype of a design to be reused in the actual design.

An evolutionary design system using a generative representation would start with a library of basic parts and would iteratively create new, more powerful modules from ones already in its library. This principle of modularity is well accepted as a general characteristic of good design since it typically promotes decoupling and reduces complexity [18]. Another advantage of reusing modules throughout a design is that a design built with a library of standard modules requires less time to verify and test because it consists of fewer unique components. Also, fewer unique components and reduced complexity should simplify manufacturing, and also leads to a smaller stockpile of spare parts necessary for maintenance and repair.

Generative representations improve the scalability of evolutionary design systems because the hierarchical reuse of modules captures some types of design dependencies and increases the ability of the search engine to navigate large design spaces. The reuse of elements in a design allows a generative representation to capture design dependencies by giving it the ability to make coordinated changes in several parts of a design simultaneously. For example, if all the legs of a robot reuse the same component, then changing the length of that component will change the length of all legs simultaneously. Navigation of large design spaces is improved through the ability to manipulate assemblies of components as units. For example, if adding/removing an assembly of m parts would make a design better, this would require the manipulation of m elements of a design encoded with a non-generative representation. With a generative representation the ability to add/remove copies of a previously created module allows for selecting these m parts in a more meaningful way than choosing them at random.

Having presented an argument for the use of generative representations, the rest of this chapter gives a detailed description of our evolutionary design system for creating robots and gives a comparison between evolution with a generative representation and a non-generative representation.

I.2 Generative Representations

Generative representations are any type of representation that allows for the creation and reuse of organizational units in a design. Within this definition there are many different methods by which reuse can be achieved. The generative representation

used here is a kind of computer language within which design-construction programs are written. This language consists of a framework for designing construction rules and a set of these rules defines a program for a design. Designs are created by compiling a design program into an assembly procedure of construction commands and then executing this assembly procedure in the module which constructs designs. Since it is a general framework for encoding designs the person using this framework must supply the set of design-construction commands and a design constructor.

A design encoded with our generative representation consists of a set of rules for constructing a design and an initial command to start the execution of this set of rules. Each rule consists of a rule head followed by a number of condition-body pairs. For example in the following rule,

$$A(n0, n1) : n1 > 5 \rightarrow B(n1{+}1)cD(n1{+}0.5, n0{-}2)$$

the rule head is A($n0$, $n1$), the condition is $n1 > 5$ and the body is B($n1$+1) c D($n1$+0.5, $n0$-2). The following is an example of a complete encoding of a design:

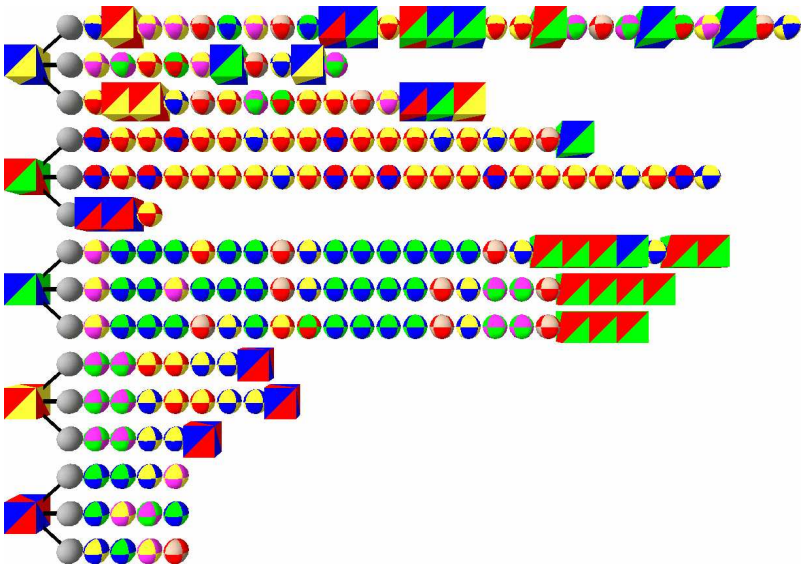
$$\begin{array}{ll} P0(4) & \\ P0(n0): & n0 > 1.0 \rightarrow [\ P1(n0 * 1.5)\ ]\ a(1)\ b(3)\ c(1)\ P0(n0-1) \\ & \\ P1(n0): & n0 > 1.0 \rightarrow \{\ [\ b(n0)\ ]\ d(1)\ \}(4) \end{array}$$

This example contains the starting command, $P0(4)$ and two rules, $P0$ and $P1$, both of which take a single argument and have a single condition-successor pair.

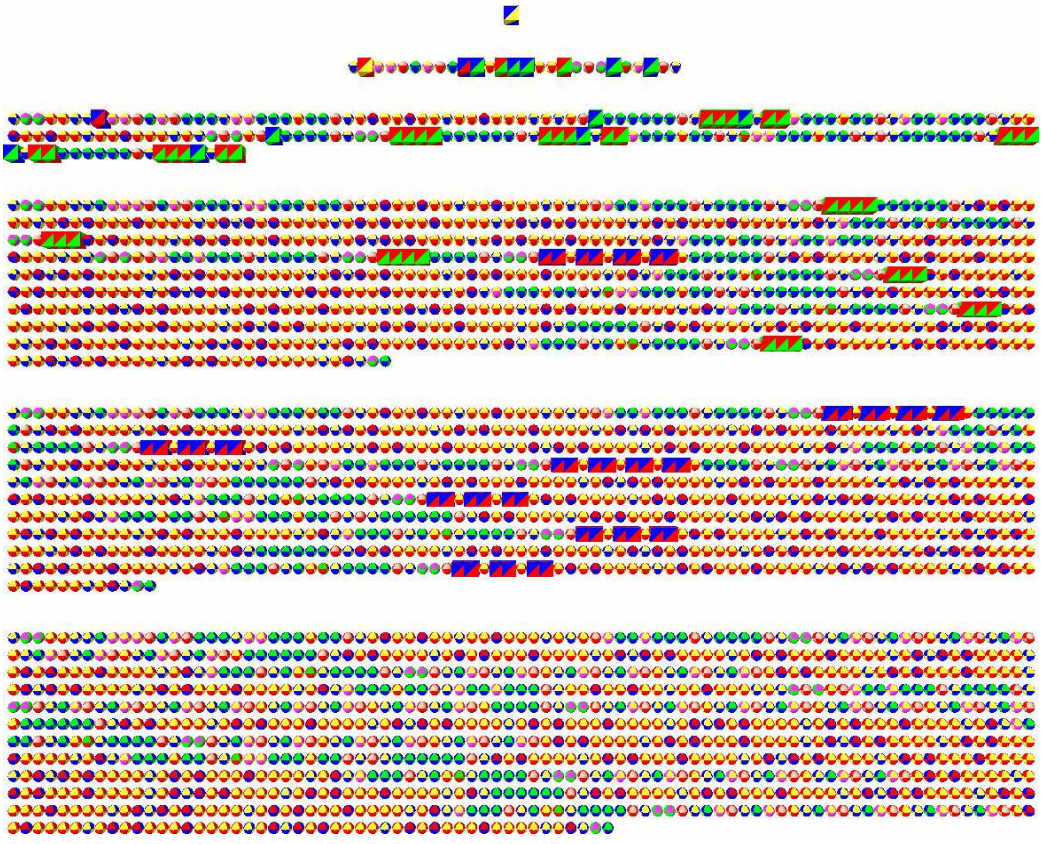
To produce an actual design from its encoding, the set of rules is first compiled into an assembly procedure. This assembly procedure is an intermediate stage between encoded design, the *genotype*, and actual design, the *phenotype*. An assembly procedure is produced by taking the starting command from the design's genotype and then iteratively rewriting rule heads with the appropriate successor. In addition to rewriting, this representation has a looping feature which replicates symbols that are enclosed within parentheses. The expression { *block* }(n) repeats the enclosed block of symbols n times and {abc}(3) compiles to, *abcabcabc*. The above program is compiled as follows:

1. *P0(4)*
2. *[P1(6)] a(1) b(3) c(1) P0(3)*
3. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [P1(4.5)] a(1) b(3) c(1) P0(2)*
4. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [P1(3)] a(1) b(3) c(1) P0(1)*
5. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [{ [b(3)] d(1) }(4)] a(1) b(3) c(1)*
6. *[[b(6)] d(1) [b(6)] d(1) [b(6)] d(1) [b(6)] d(1)] a(1) b(3) c(1) [[b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1)] a(1) b(3) c(1) [[b(3)] d(1) [b(3)] d(1) [b(3)] d(1) [b(3)] d(1)] a(1) b(3) c(1) b(3)*

A program for encoding a design can be shown graphically by representing the different parts and symbols of a program with different shapes and colors. The



(a)



(b)

Figure I.1. Graphical version of the generative representation, (a); along with the sequence of strings produced, (b).

images in figure I.1 show a larger set of rules for a design, figure I.1.a, as well as the sequence of assembly procedures that are generated in the compilation process, figure I.1.b. In these images rule-head symbols are represented by cubes with lines connecting them to their condition-body pairs, grey spheres represent the condition and the symbols following it are the body. The sequence is started with the first cube (here a blue and yellow one) and the sequence of symbols below it are the assembly procedures generated after each iteration of parallel replacement.

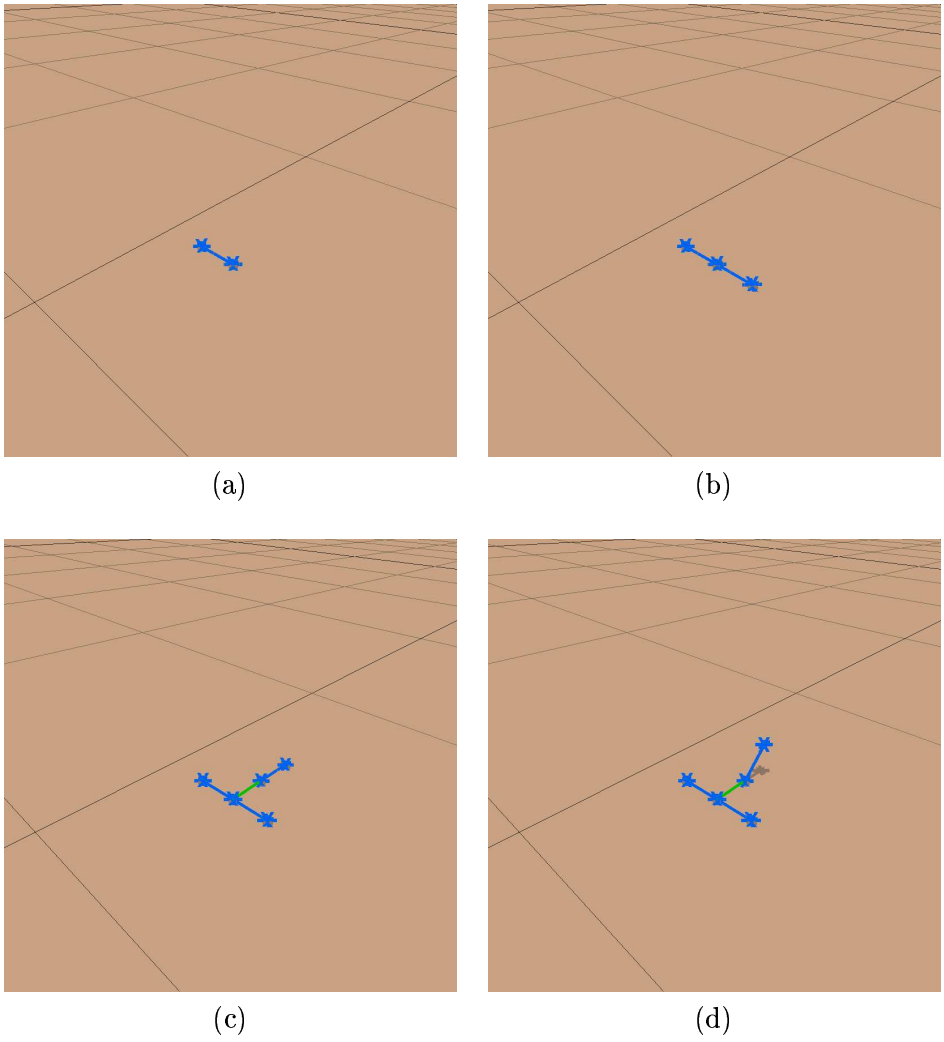
I.3 Encoding Robots with a Generative Representation

To create designs with the generative representation of the previous section, the non-rule-head symbols are interpreted as construction commands in a design construction language. The class of robots which we evolve consist of Tinker-Toy^{TM}-like rods of regular length and both fixed and actuated joints. Since these robots are evolved using a generative representation we call them *genobots*, for *g*eneratively *en*coded r*obots*. For these robots there are two types of controllers which are used to drive them: simple oscillators, and neural-networks.

I.3.1 Oscillator Controlled Genobots

The command set for constructing oscillator-controlled genobots consists of operators for attaching fixed rods and actuated rods as well as operators for controlling the oscillation of the actuated rods. To know where to add the next part the robot-construction module maintains the current construction state, which consists of the current position and orientation on the robot body and also the relative phase offset for oscillators. The commands *turn-left/right/up/down/clockwise/counter-clockwise(n)* rotate the current heading about the appropriate axis in units of 90°. Rods are added to the robot with the command *forward*, which adds a rod in the forward direction if none exists or moves the current position to the end of the rod if one does exist. The command *back* moves the current position back to the other end of the current rod. Actuated rods are added with a set of commands which specify the joint type to use and its oscillation rate. *Revolute-1(n)* creates a joint which oscillates from 0° to 90° about the Z-axis with speed n, *revolute-2(n)* creates a joint which oscillates from $-45°$ to 45° about the Z-axis with speed n, *twist-90(n)* creates a joint which oscillates from 0° to 90° about the X-axis with speed n, and *twist-180(n)* creates a joint which oscillates from $-90°$ to 90° about the X-axis with speed n. These actuated-rod commands specify the rate of oscillation, to control the relative phase of the oscillating joints the construction state maintains an offset value that is assigned to newly created actuated joints. This phase-offset value is changed with the commands `increase-offset(n)` and `decrease-offset(n)`, which adjust the phase-offset value in increments of 25%. To allow for a kind of branching in executing robot-construction assembly procedures there are commands, `[` and `]`, for pushing and popping the construction state to/from a stack.

Figure I.2 contains images of intermediate steps in building a genobot, as well as part of its animation, from the following command sequence,



(a) (b) (c) (d)

Figure I.2. Building and simulating a three-dimensional robot.

[left(1) forward] [right(1) forward] revolute-1(1) forward

The single bar in figure I.2.a is built from the string, *[left(1) forward]*, and the two bar structure in figure I.2.b is built from, *[left(1) forward] [right(1) forward]*. The final robot is made from the command sequence, *[left(1) forward] [right(1) forward] revolute-1(1) forward*, and is shown in figure I.2.c, where it is displayed part-way through its movement cycle. Figure I.2.d displays the robot with the actuated joint moved half-way through its joint range.

I.3.2 Neural Networks

The method for constructing the neural-network controllers starts with a single node and edge and adds new nodes/edges through the execution of network-construction commands. Commands for constructing the network operate on links between neurons and use the most recently created link as the current one. *Push* and *pop* operators, '[' and ']', are used to store and retrieve the current link – consisting of the from-neuron, the to-neuron and index of the link into the to-neuron (for when there are multiple links between neurons) – to and from the stack. This stack of edges allows a form of branching to occur in an encoding – an edge can be pushed onto the stack followed by a sequence of commands and then a pop command makes the original edge the current edge again. For the following list of commands the current link connects from neuron A to neuron B.

- decrease-weight(n) – Subtracts n from the weight of the current link. If the current link is a virtual link, it creates it with weight $-n$.
- duplicate(n) – Creates a new link from neuron A to neuron B with weight n.
- increase-weight(n) – Add n to the weight of the current link. If the current link is a virtual link, it creates it with weight n.
- loop(n) – Creates a new link from neuron B to itself with weight n.
- merge(n) – Merges neuron A into neuron B by copying all inputs of A as inputs to B and replacing all occurrences of neuron A as an input with neuron B. The current link then becomes the nth input into neuron B.
- next(n) – Changes the from-neuron in the current link to its nth sibling.
- output(n) – Creates an output-neuron, with a linear transfer function, from the current from-neuron with weight n. The current-link continues to be from neuron A to neuron B.
- parent(n) – Changes the from-neuron in the current link to the nth input-neuron of the current from-neuron. Often there will not be an actual link between the new from-neuron and to-neuron, in which case a virtual link of weight 0 is used.
- reverse – Deletes the current link and replaces it with a link from B to A with the same weight as the original.

- set-function(n) – Changes the transfer function of the to-neuron in the current link, B, with: 0, for sigmoid; 1, linear; and 2, for oscillator.

- split(n) – Creates a new neuron, C, with a sigmoid transfer function, and moves the current link from A to C and creates a new link connecting from neuron C to neuron B with weight n.

Neurons in the network are initialized to an output value of 0.0 and are updated sequentially by applying a transfer function to the weighted sum of their inputs with their outputs clipped to the range ± 1. The different transfer functions are: sigmoid, using $tanh(sum\ of\ inputs)$; linear; and an oscillator. Oscillator units maintain a state which is increased by 0.01 after each update. The output of an oscillator unit is mapped to the range -1 to 1 by applying a triangle wave function, with a period of four, to the sum of its inputs and its state. The initial activation value for neurons with the sigmoid and linear transfer functions is 0.0 and the initial activation value for oscillator units is 1.0.

I.3.3 Neural-Network Controlled Genobots

A genobot's morphology and neural controller are constructed by combining the command sets for constructing body and brain into one language and then building body and brain simultaneously. Since actuated joints are controlled by the neural network the commands for modifying the phase offset, `increase-offset(n)` and `decrease-offset(n)`, are not included. The resulting design language consists of the commands: `[`, `]`, `(`, `)`, `forward`, `back`, `revolute-1`, `revolute-2`, `twist-90`, `twist-180`, `left(n)`, `right(n)`, `up(n)`, `down(n)`, `clockwise(n)`, `counter-clockwise(n)`, `decrease-weight(n)`, `duplicate(n)`, `increase-weight(n)`, `loop(n)`, `merge(n)`, `next(n)`, `parent(n)`, `reverse`, `set-function(n)`, and `split(n)`. This language has two push/pop commands with two stacks: ( ), for pushing/popping the link-state to the link stack; and [ ], for pushing/popping both the morphology and link states to a stack. A robot's body and brain are joined together by attaching the current input-neuron to the newly created actuated joint each time a joint command – *revolute-1*, *revolute-2*, *twist-90*, or *twist-180* – is executed. By defining joint-creation commands in a way that affects both controller and morphology a connection between body and brain is induced.

An example of an assembly procedure using this language is,

[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward] duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4) reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward

Figure I.3 contains a sequence of images showing intermediate stages in the construction of this assembly procedure. Before any commands are processed a robot consists of a single oscillating neuron and a point, figure I.3.a. After executing the commands, *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0)*

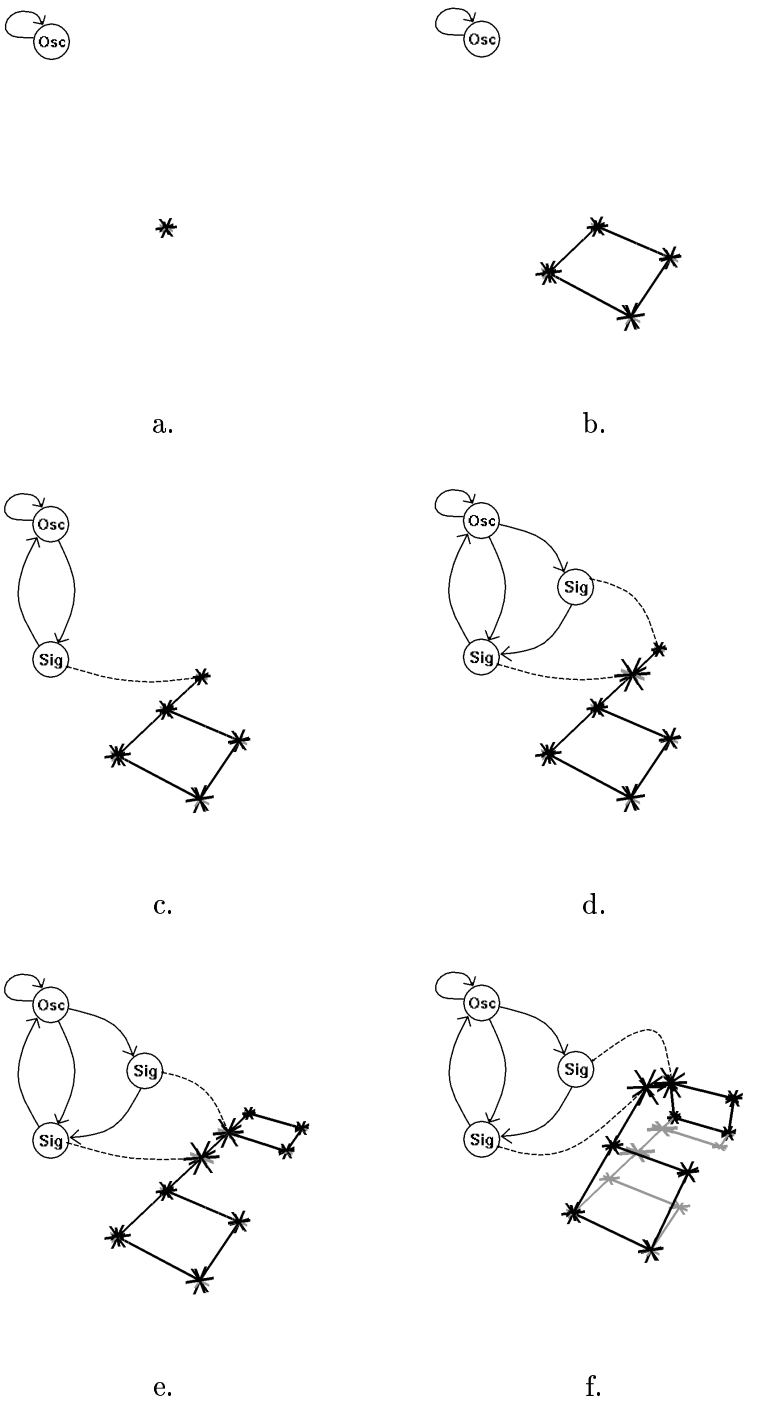


Figure I.3. Constructing a neural-network controlled genobot

forward], the robot consists of a square of four rods and the oscillating neuron, figure I.3.b. After executing, *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, a second neuron is created and it is attached to the actuated joint at the end of the newly created rod, figure I.3.c. The commands, *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, are repeated and a third neuron is created and it is attached to another actuated joint, figure I.3.d. The last commands, *left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*, attach another square onto the end of the last revolute-1 joint, figure I.3.e. Figure I.3.f shows the genobot with the joints halfway through their movement range.

I.3.4 Generative Representation Example for Neural-Network Controlled Genobots

Having presented construction languages for oscillator-controlled robots, neural networks and neural-network controlled robots we now give an example going from genotype to phenotype of a neural-network controlled robot. The encoding for this genobot consists of the starting command $P0(4)$, two production rules, each with two condition-successor pairs, and uses the command set for constructing neural-network controlled genobots:

$$
\begin{array}{lll}
P0(4) & & \\
P0(n0): & n0 > 3.0 \rightarrow & P1(5.0)\ P0(n0-2.0)\ left(1.0)\ P1(4.0) \\
 & n0 > 0.0 \rightarrow & \{\ duplicate(0.25)\ split(0.4)\ reverse \\
 & & revolute\text{-}1(1.0)\ \}(2.0) \\
 & & \\
P1(n0): & n0 > 4.0 \rightarrow & [\ P1(4.0)\ ] \\
 & n0 > 0.0 \rightarrow & \{\ right(1.0)\ forward\ \}(n0)
\end{array}
$$

To produce the assembly procedure for constructing the genobot the rule-system is compiled starting with the command *P0(4)*:

1. *P0(4)*
2. *P1(5.0) P0(2.0) left(1.0) P1(4.0)*
3. *[P1(4.0)] { duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0) { right(1.0) forward }(4.0)*
4. *[{ right(1.0) forward }(4.0)] { duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0) { right(1.0) forward }(4.0)*
5. *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward] duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4) reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*

This last sequence of commands is the assembly procedure from the example in section I.3.3 and produces the genobot in figure I.3.f.

I.4 Robot Simulator

Once an assembly procedure for constructing a robot has been executed and the resulting robot is constructed, its behavior is evaluated in a quasi-static, kinematics simulator. The kinematics are simulated by computing successive frames of moving joints in small angular increments of 0.001 radians toward the desired angle. This angle is determined by either an oscillator or a neuron. Oscillators cycle between -1 and 1 and this is scaled to the joint's range of motion. Similarly, the output value of a neuron falls within the range of -1 and 1 and this is scaled to the joint's range of motion. After each update the structure is then settled by determining whether or not the robot's center of mass falls outside its footprint and then repeatedly rotating the entire structure about the edge of the footprint nearest the center of mass until it is stable.

To achieve robot designs that are robust to transferal to the real world, noise is added to evolved structures similar to the method of [9] and [7]. A robot design is evaluated by simulating it three times, once without noise and twice with different amounts of construction noise applied to joint angles. Noise is applied to all connections that are not part of a cycle and is a random rotation in the range of ± 0.1 radians about each of the three coordinate axis. The returned fitness of an evolved individual is the worst fitness score from the three trials. By adding construction noise to a robot and evaluating it multiple times with different random noise each time, evolved robots are made robust to imperfections in real-world construction.

I.5 Evolutionary Design System

To demonstrate the advantages of generative representations we use GENRE, an evolutionary design system for creating designs [4]. GENRE consists of several design constructors and fitness functions, the compiler for the generative representation and an evolutionary algorithm (EA) for searching the design spaces. The EA is the module that drives GENRE and it operates by processing a population of designs (members of which are called *individuals*) encoded with the generative representation. Search is started by creating an initial random population of individuals and evaluating each of these with a user-defined *fitness function*, a mathematical expression for scoring the goodness of a design. The EA then creates successive new populations by selecting the better individuals of the current population and applying small amounts of variation to their encoding to produce new individuals in a new population.

The two variation operators that are used to produce new individuals are *mutation* and *recombination*. Mutation creates a new individual by copying the parent individual and making a small change to it, such as by replacing one command with another, perturbing the parameter in a command by adding/subtracting a small value to it, or adding/deleting a sequence of commands in a rule body. Recombination takes two individuals as parents and creates a new individual by making a copy of the first parent and then either exchanging a rule with the second parent, or randomly replacing a sequence of commands in one body with a sequence from the second parent.

Designs can be encoded with either the generative representation described in section I.2 or a non-generative representation. For designs encoded with the generative representation the design program is first compiled into a sequence of construction commands called an assembly procedure. This assembly procedure is then executed by the design constructor to produce the encoded design. For the non-generative representation each individual in the population is an assembly procedure which specifies how to construct the design. We implement this assembly procedure as a degenerate version of the generative representation which has only a single rule in which the condition always succeeds and the body consists only of construction commands. Implementing the non-generative representation in the same way as the generative representation allows us to use the same evolutionary algorithm and the same variation operators; the only difference between the two representations is the ability to hierarchically reuse elements of encoded designs. Once a design has been constructed, using either the generative or non-generative representation, it is evaluated for how good it is with the user-defined fitness function.

I.6 Evolution of Oscillator Controlled Robots

To demonstrate the advantages of generative representations for robot design we first compare the generative representation of section I.2 against the non-generative representation described in section I.5 for the evolution of oscillator controlled robots. For this comparison the non-generative representation is implemented as a degenerate type of generative representation with one production rule, no arguments, one condition-successor pair whose condition always succeeds, and without the repeat operator or the ability to call production rules. The maximum length of the production body is set to ten thousand symbols, allowing assembly procedures of up to ten thousand operators to be evolved. The generative representation has fifteen production rules, three condition-successor pairs, and two parameters for each production rule. For the generative representation, the maximum length of production body is set to fifteen commands and the maximum allowed length of a compiled assembly procedure is set to ten thousand operators – the same length as with the non-generative representation. The evolutionary algorithm used a population of one hundred individuals and was run for five hundred generations and results are the average over ten trials.

The design problem for this comparison is to produce robots that move across the ground as fast as possible with fitness being a function of the distance moved by the robot's center of mass. In order to discourage sliding, fitness was reduced by the distance that points of the robot's body were dragged along the ground. Since the kinematics simulator detects collisions but cannot handle them like a physical dynamics simulator would, robots that have collisions between body parts are given a fitness of zero. Finally, a robot was given zero fitness if it had a sequence of four or more rods in which none of the rods was part of a closed loop with other rods. This constraint was intended to keep the system from producing spindly robots which would not function well in reality. The graph in figure I.4 plots the fitness of the best individual in the population, averaged over ten trials, for both the non-generative

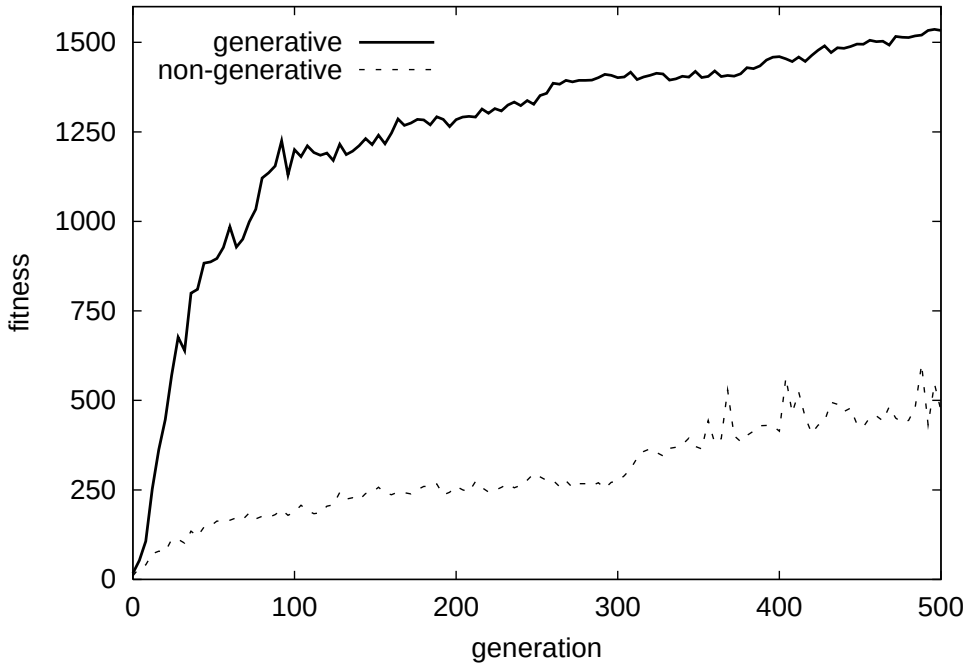
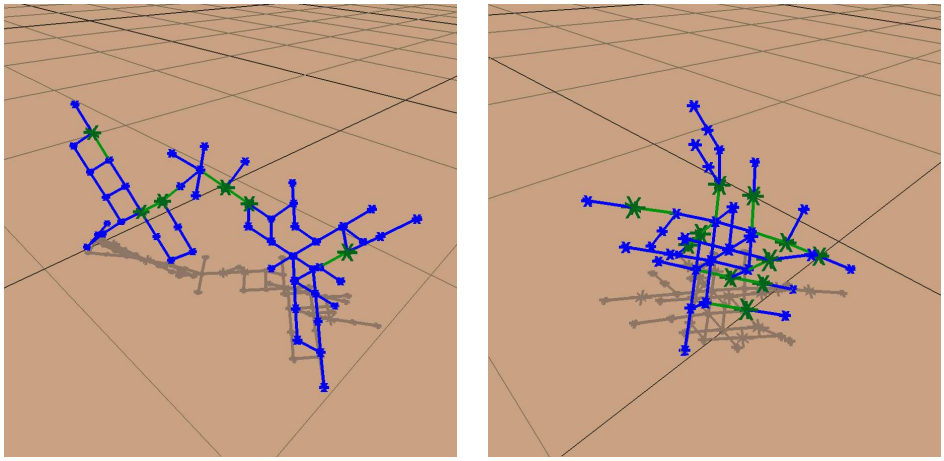


Figure I.4. Performance comparison between the non-generative and generative representations on evolving robots with oscillator networks for controllers.

and generative representations. This graph shows that far better robots are evolved with the generative representation (with an average best fitness of just over fifteen hundred) than with the non-generative representation (with an average best fitness of approximately five hundred).



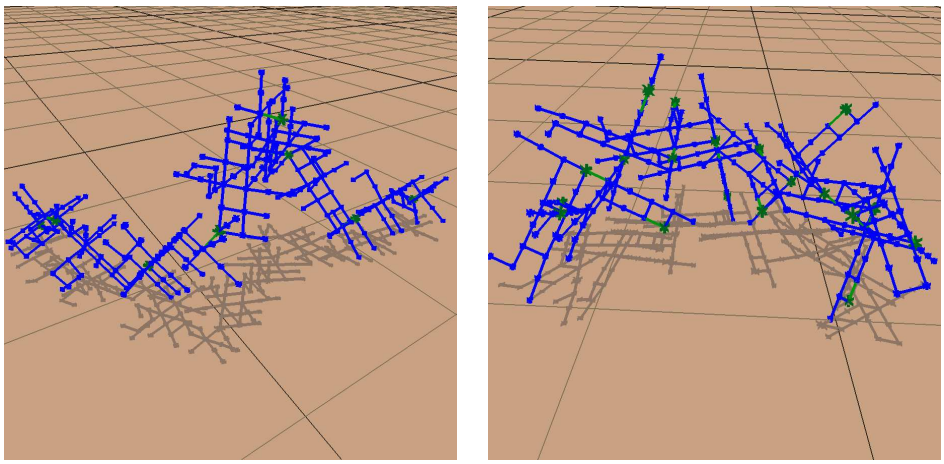
(a) (b)

Figure I.5. The two best oscillator controlled robots evolved using the non-generative representation.

Robots evolved with the non-generative representation tended to have few parts, moved awkwardly and had little regularity in their structure. The two main forms of locomotion found were using one or more appendages to push along or having two main body parts connected by a sequence of rods that twisted in such a way that first one half of the robot would rotate forward, then the other. The two fastest robots evolved with the non-generative representation are shown in figure I.5.a (fitness 1188 with 49 rods and moves by twisting) and figure I.5.b (fitness 1000 with 31 rods which moves by pushing).

Genobots evolved with the generative representation not only had higher average fitness, but tended to move in a more continuous manner. As with evolution using the non-generative representation, the genobots created at the beginning of an evolutionary run had a few rods and joints that would slowly slide on the ground. In most evolutionary runs, faster genobots were evolved by repeating rolling segments to smoothen out gaits or by increasing the size of segments/appendages to increase the distance moved in each oscillation period. Of these, the two fastest are the genobot in figure I.6.a, whose segments are shaped like a coil and it moves by rolling sideways with fitness of 3604 and 325 rods, and the genobot in figure I.6.b, a sequence of interlocking X's that rolls along with fitness 2754 and 268 rods. An example of the movement cycle of a genobot produced with the generative representation is in figure I.7.

While additional runs with the non-generative representation failed to produce designs more interesting than those in figure I.5, additional runs with the generative



(a) (b)

Figure I.6. The two best oscillator controlled genobots evolved using the generative representation.

representation produced a variety of genobots with different styles of locomotion. The most common form of movement for evolved oscillator-controlled genobots was to roll along sideways, as done by the chains in I.8.a and I.8.b. The genobot in I.8.c moves like an undulating sea-serpent. One of the larger genobots that evolved is the one in figure I.8.d which uses four legs in an awkward walk.

I.7 Evolution of Neural-Network Controlled Robots

Next we compare the generative representation against the non-generative representation on evolving neural-network controlled robots. The same fitness function, a measure of distance travelled, is used as with the experiments for oscillator-controlled robots and both the generative and non-generative representation are configured in the same way. The results for this comparison are shown in figure I.9, which contains a graph of the fitness (averaged over ten trials) of the best individuals evolved with the non-generative representation and the generative representation. After ten generations the generative representation achieved a higher average fitness than runs with the non-generative representation did after 250 generations and the final genobots evolved with the generative representation were on average more than ten times faster than robots evolved with the non-generative representation.

Figure I.10 shows the two best individuals evolved with the non-generative representation and figure I.11 shows the two best genobots evolved with the generative representation. From the images it can be seen that the robots evolved with the non-generative representation are irregular and have few components, whereas the genobots evolved with the generative representation are more regular and, in some cases, have two or more levels of reused assemblies of components.

As with the oscillator-controlled robots, further runs with the generative representation produced robots with different styles of locomotion but failed to produce

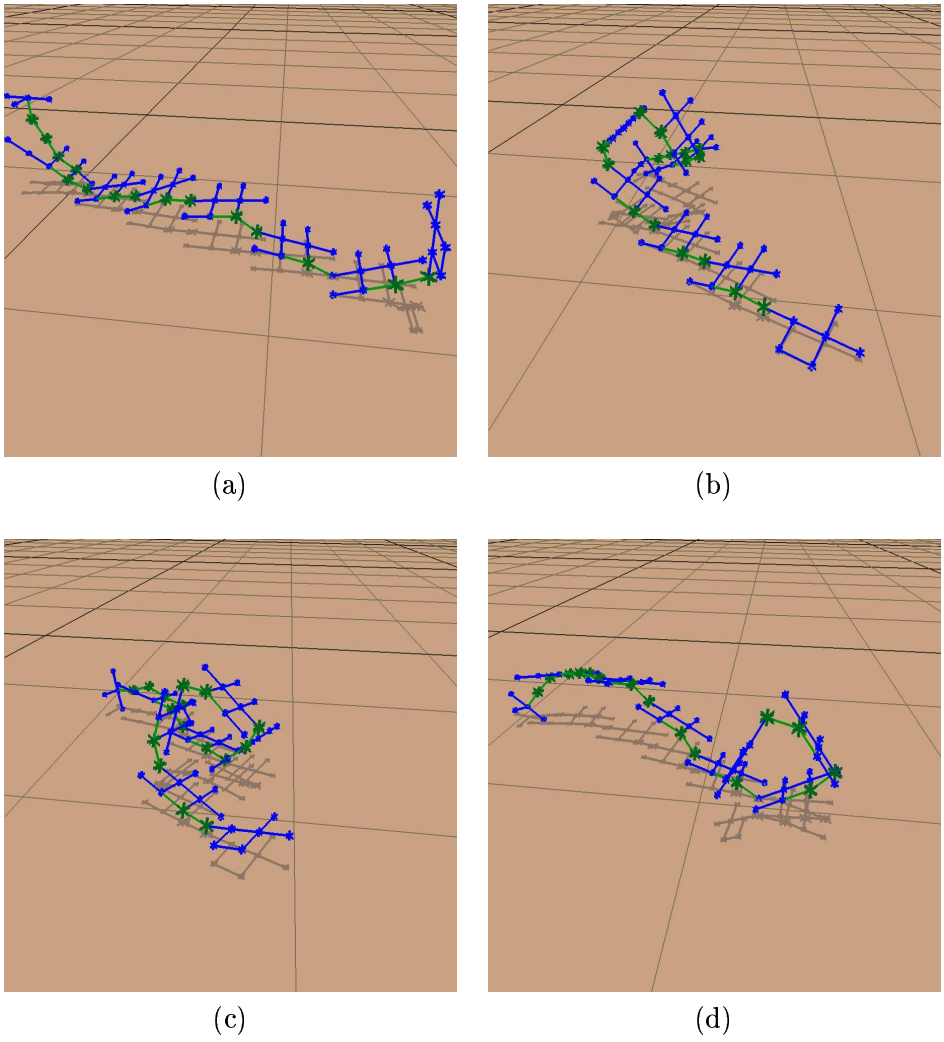


Figure I.7. Part of the locomotion cycle of an oscillator-controlled genobot.

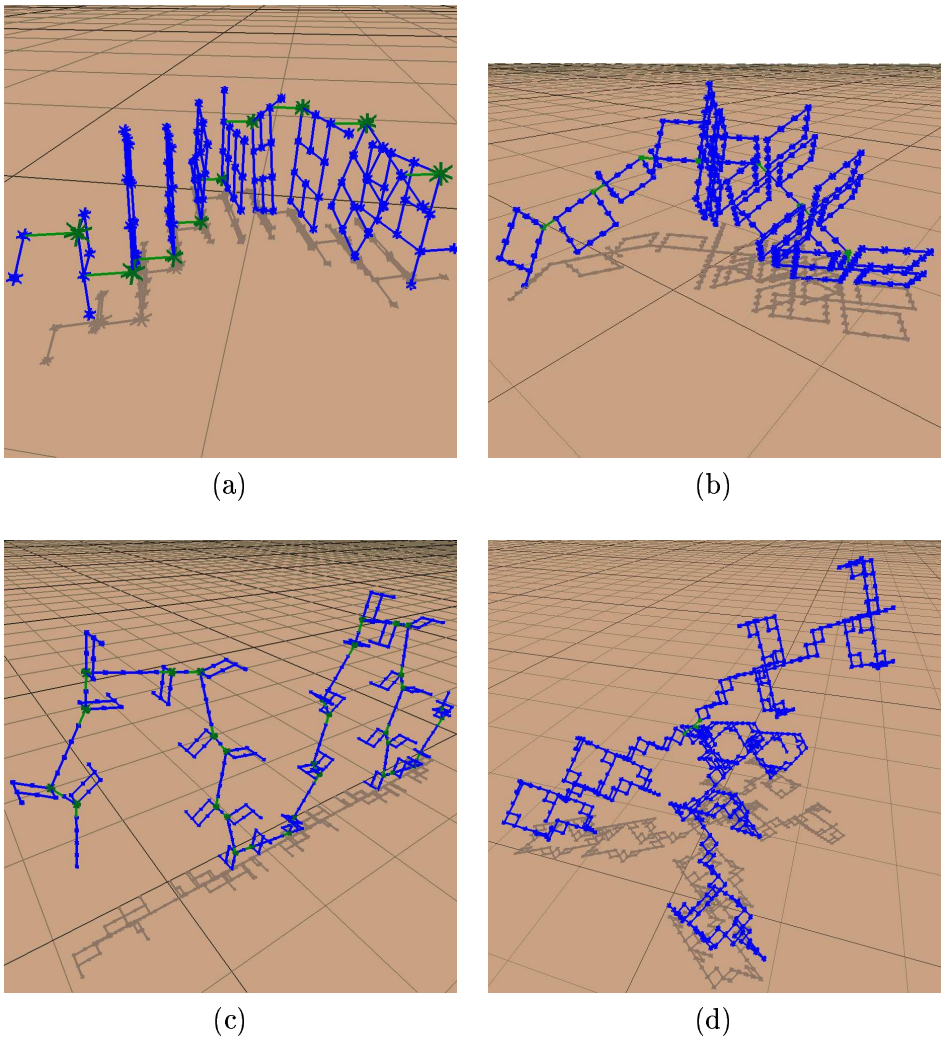


Figure I.8. A variety of evolved 3D oscillator robots.
These consist of: *a*, a sequence of rolling rectangles with 169 bars; *b*, an asymmetric rolling genobot with 306 bars; *c*, an undulating serpent with 339 bars; *d*, a four-legged walking genobot with 629 bars.

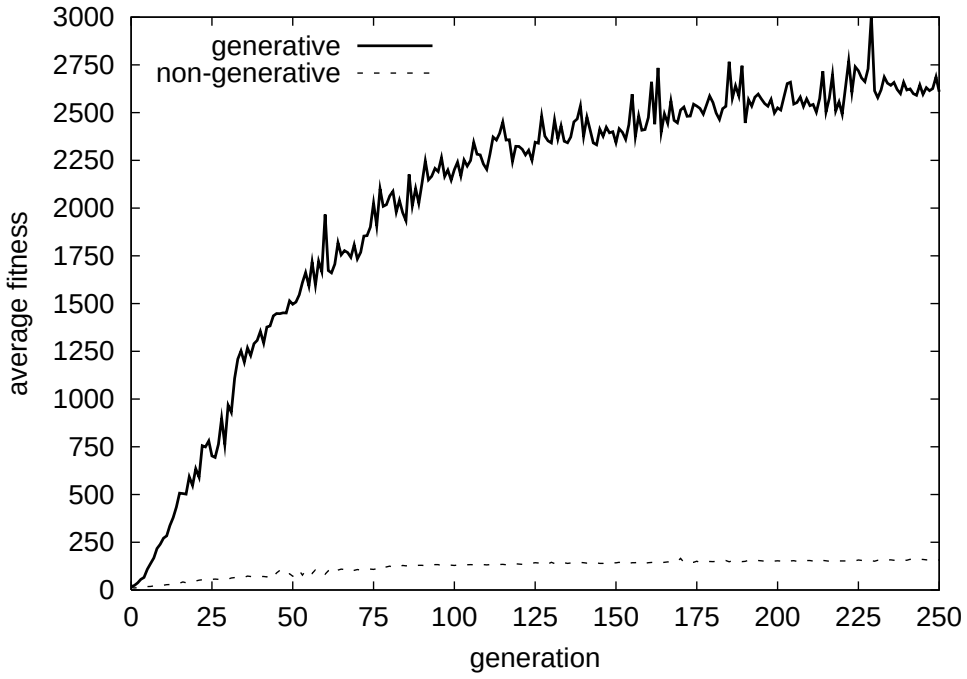
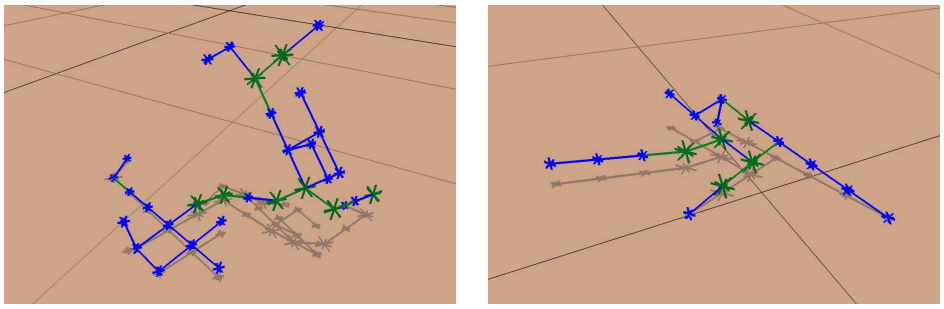


Figure I.9. Performance comparison between the non-generative representation and the generative representation on evolving robots with neural networks for controllers.

new varieties of robots with the non-generative representation. The images in figure I.12 are examples of other neural-network controlled robots evolved with the generative representation. The genobot in figure I.12.a is comprised almost entirely of actuated joints and moves by alternating between pulling all its limbs in tight to its body and extending them while twisting its torso. In another evolutionary run a wheel-like genobot was evolved that moves by using its tail to continually turn its body over and over, figure I.12.b. The robot in figure I.12.c has articulated joints between body segments for a kind of inchworm-like motion and is similar to early versions of the undulating serpent of figure I.8.c. Finally, the robot in figure I.12.d is an example of a rolling chain of segments which is the method of locomotion most commonly evolved for both oscillator and neural-network controlled genobots.

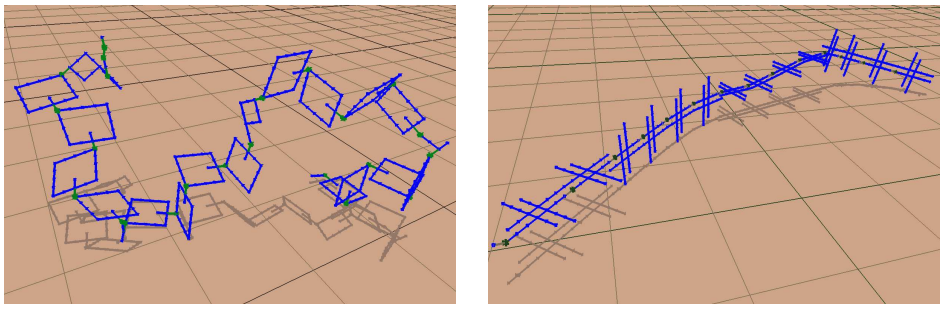
I.8 Advantages of a Generative Representation

The central claim of this chapter is that using generative representations improves the evolvability of designs by capturing design dependencies and improving the ability of the search algorithm to navigate through large design spaces in a meaningful way. This can be intuitively understood by looking at some examples of robots evolved with a generative representation.



(a) (b)

Figure I.10. The two best neural-network controlled robots evolved with the non-generative representation.



(a) (b)

Figure I.11. The two best neural-network controlled genobots evolved with the generative representation.

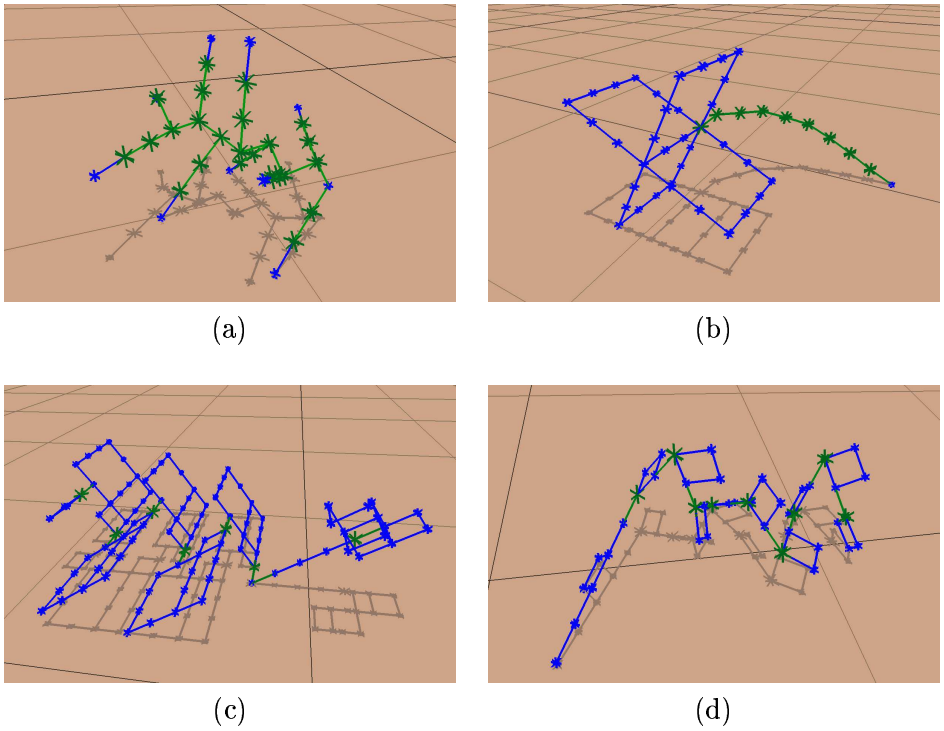
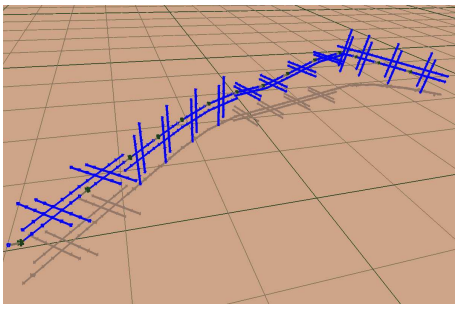
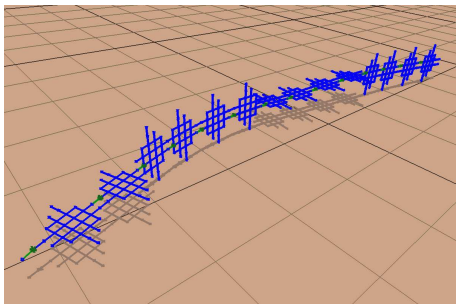


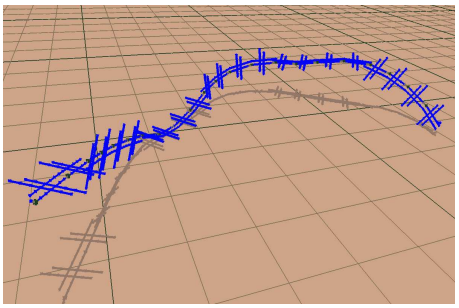
Figure I.12. Other genobots evolved using the generative representation on runs with no constraints on limb lengths.



(a)



(b)



(c)

Figure I.13. Mutations of a genobot: (a), the genobot from figure I.11.b; (b), a change to a low-level component of parts results in all occurrences of this part to have the change; (c), a single change to the genotype changes the number of high-level components in the genobot from four to six.

An advantage of the design encodings that are found with a generative representation is in the changes that can be made to this encoding that can not be made to designs encoded with a non-generative representation. The images in figure I.13 are an example which shows the benefits of reuse through variations applied to the genobot in figure I.11.b. Changing the genotype to add rods to an assembly of parts results in the change to all occurrences of that part in the design, figure I.13.b, and a single change to the genotype can cause the addition/subtraction of a large number of parts, figure I.13.c. With a non-generative representation, these changes would require the simultaneous changes of multiple symbols in the encoding. Some of these changes must be done simultaneously for the resulting design to be viable – changing the height of only one leg of a walking robot can result in a significant loss of fitness – and so these changes are not evolvable with a non-generative representation. Others, such as producing a new body-segment, are viable with a series of single-rod changes. Yet, in the general case this could result in a significantly slower search speed in comparison with a single change to a robot encoded with a generative representation.

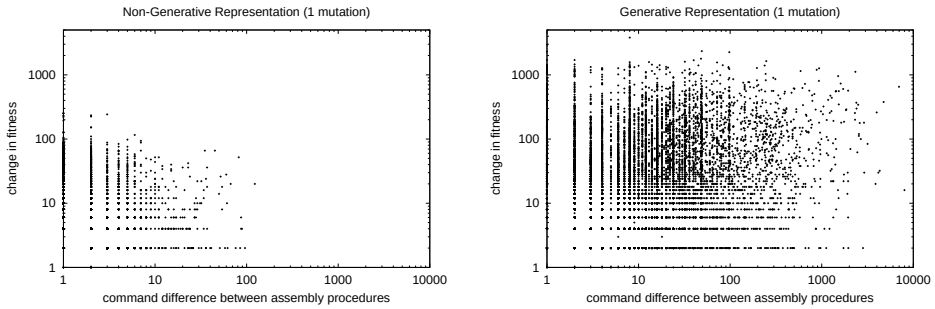
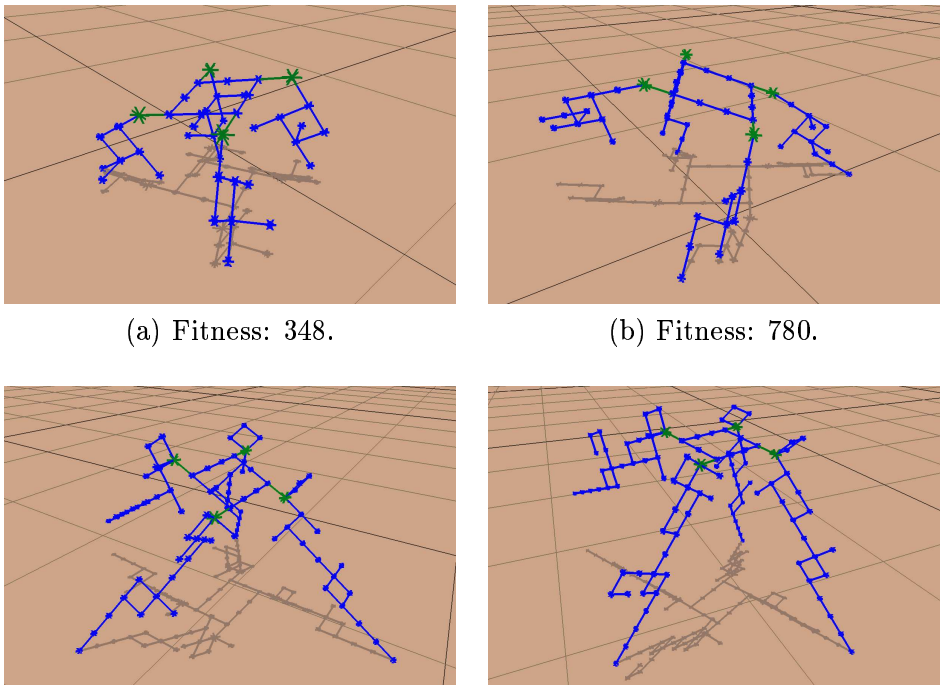


Figure I.14. Plot of amount of change in assembly procedures from parent to child versus change in fitness for trials evolving robots.

The graphs in figure I.14 are scatter plots of the command difference between a parent and child's assembly procedures against their change in fitness on the robot design problem. These graphs show that as the size of change in the resulting design increases it is more likely to be an improvement on designs encoded with a generative representation than those encoded with a non-generative representation. This means that search algorithms are better able to use large movements in the design space to navigate through the design space with the generative representation.

That the evolutionary design system is taking advantage of the ability to make coordinated changes with a generative representation is demonstrated by individuals taken from different generations of the evolutionary process. The sequence of images in figure I.15, which are of the best individual in the population taken from different generations, show two changes occurring. First, the rectangle that forms the body of the genobot goes from two-by-two (figure I.15.a), to three-by-three (figure I.15.b), before settling on two-by-three (figures I.15.c-d). These changes are possible with



(a) Fitness: 348. (b) Fitness: 780.

(c) Fitness: 1450. (d) Fitness: 2192.

Figure I.15. Evolution of a four-legged walking genobot.

a single change on a generative representation but cannot be done with a single change on a non-generative representation. The second change is the evolution of the genobot's legs. That all four legs are the same in all four images strongly suggests that the same module in the encoding is being used to create them. As with the body, changing all four legs simultaneously can be done easily with the generative representation by changing the one module that constructs them, but would require simultaneously making the same change to all four occurrences of the leg assembly procedure in the non-generative representation.

One other advantage of using a generative representation is that by encoding an object through a set of reusable rules for constructing it, it is possible to encode a class of designs. By evaluating an individual with different parameters to its starting command families of designs can be evolved, such as the tables in figure I.16 [5].

I.9 Summary

Only in the last few years has the computer-automated design of robots succeeded in transferring designs that are produced in simulation to the real-world [12]. The next challenge has been in creating such systems that can produce designs of complexities comparable to real-world robots. Existing systems for designing robots have been limited by the representation they use to encode designs. As with both traditional engineering and software design, for evolutionary design systems to be able to scale

(a) (b)

Figure I.16. Two tables from a family of designs.

they must be able to hierarchically reuse modules throughout a design. For designs to be able to reuse modules the representation scheme for encoding them must allow it.

Here we have defined generative representations as the class of representations which allow modules encoded in the genotype to be reused in producing the actual design and have claimed that evolution with generative representations will improve scalability. To support this claim we described a generative representation and an automated design system for creating robots. Using this system we evolved oscillator-controlled and neural-network controlled robots for a locomotion task using both a non-generative and a generative representation. The results of this comparison showed that better robots were evolved with the generative representation than with the non-generative representation, and that robots encoded with the generative representation had a modular structure wth a reuse of assemblies of components.

This comparison has shown how important the representation is for an evolutionary design system to scale to complex, high-part count designs. The next step in automated design is in producing design representations that can hierarchically create and reuse assemblies of parts in ever more powerful ways. As continuing work expands the range and power of generative representations, while maintaining evolvability, we expect to see ever more progress toward general purpose evolutionary design systems.

Acknowledgements

Images and graphs are copyright Gregory S. Hornby and used with permission.

Bibliography

[1] P. J. Bentley, editor. *Evolutionary Design by Computers.* Morgan Kaufmann, San Francisco, 1999.

[2] P. J. Bentley and D. W. Corne, editors. *Creative Evolutionary Systems.* Morgan Kaufmann, San Francisco, 2001.

[3] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 63–70, Los Angeles, California, August 1995. In *Computer Graphics* Annual Conf. Series, 1995.

[4] G. S. Hornby. *Generative Representations for Evolutionary Design Automation.* PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA, 2003.

[5] G. S. Hornby. Generative representations for evolving families of designs. In E. Cantu-Paz et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference*, LNCS 2724, pages 1678–1689, Berlin, 2003. Springer-Verlag.

[6] G. S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous evolution of gaits with the sony quadruped robot. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1297–1304. Morgan Kaufmann, 1999.

[7] G. S. Hornby, S. Takamura, O. Hanagata, M. Fujita, and J. Pollack. Evolution of controllers from a high-level simulator to a high dof robot. In J. Miller, editor, *Evolvable Systems: from biology to hardware; Proc. of the Third Intl. Conf.*, Lecture Notes in Computer Science; Vol. 1801, pages 80–89. Springer, 2000.

[8] C. C. Huang and A. Kusiak. Modularity in design of products and systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 28(1):66–77, 1998.

[9] N. Jakobi. *Minimal Simulations for Evolutionary Robotics.* PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, May 1998.

[10] M. Komosinski. The world of framsticks: Simulation, evolution, interaction. In *Virtual Worlds 2*, Lecture Notes in Artificial Intelligence 1834, pages 214–224. Springer-Verlag, 2000.

[11] D. S. Linden. Innovative antenna design using genetic algorithms. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 20, pages 487–510. Morgan Kaufmann, San Francisco, 2001.

[12] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[13] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[14] Z. Michalewicz, D. Dasgupta, R. G. Le Riche, and M. Schoenauer. Evolutionary algorithms for constrained engineering problems. *Computers and Industrial Engineering Journal*, 30(2):851–870, 1996.

[15] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93 Conference Proceedings*, pages 343–350, 1993. Annual Conference Series.

[16] G. Robinson, M. El-Beltagy, and A. Keane. Optimization in mechanical design. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 6, pages 147–165. Morgan Kaufmann, San Francisco, 1999.

[17] K. Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, pages 28–39, Boston, MA, 1994. MIT Press.

[18] N. P. Suh. *The Principles of Design.* Oxford University Press, 1990.

[19] K. Ulrich and K. Tung. Fundamentals of product modularity. *Issues in Design/Manufacture Integration - 1991 American Society of Mechanical Engineers, Design Engineering Division (Publication) DE*, 39:73–79, 1991.

[20] M. van de Panne and E. Fiume. Sensor-actuator Networks. In *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 335–342, 1993.

[21] J. Ventrella. Explorations in the emergence of morphology and locomotion behavior in animated characters. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, Boston, MA, 1994. MIT Press.